A System for Measuring Function Points

Evelina Lamma[1], Paola Mello[2], Fabrizio Riguzzi[1]

[1]DEIS - Università di Bologna
Viale Risorgimento, 2 40136 Bologna
{elamma,friguzzi}@deis.unibo.it

[2]Dipartimento di Ingegneria, Università di Ferrara
Via Saragat, 1 44100 Ferrara
pmello@ing.unife.it

Abstract: We present the system FUN for measuring the Function Point software metric from specifications expressed in the form of an Entity Relationship (ER) diagram and a Data Flow Diagram (DFD). As a first step towards the implementation of the system, the informal Function Point counting rules have been translated into rigorous rules expressing properties of the ER-DFD. Prolog was chosen for the implementation because of its declarativity and maintainability. Thanks to its relational representation, it was possible to directly represent the input ER-DFD with Prolog facts. Declarativity allowed a straightforward translation of the rigorous rules into code and a quick implementation of a working prototype. Finally, maintainability was a primary concern since the Function Point counting method is continually evolving.

1 Introduction

Software metrics are emerging as a powerful tool for the management of the software development process. Software metrics allow to apply engineering principles to software development, providing a quantitative and objective base for process and technology decisions.

Among software metrics, Function Points [Albrecht 79] [Albrecht and Gaffney 83] (FP for short in the following) give a measure of the size of the system by measuring the functionalities that the system offers the user. This metric is applicable both at the beginning of the development process, in the requirements or specification phases, and at the end of the process, after implementation. [Albrecht and Gaffney 83] showed that FP are highly correlated with work-hours and argued that FP can be used effectively for estimating effort and therefore cost. The International Function Point User Group (IFPUG) sets the international standard for FP counting rules by publishing the Function Point Counting Practices Manual [IFPUG 94a].

FP have been designed with the aim of providing a metric that is independent from both implementation techniques and from the person doing the count. The result of the count should be independent from whether it is performed from specifications or from the final system (provided that requirements have not changed), from the language, platform or software architecture employed. For this reasons, FP are gaining an increasingly widespread application in industry, where they are used to evaluate costs, both “a priori” and “a posteriori”, and for comparing different projects inside the same company or in different companies.

A number of tools [Mendes et al. 96] exist on the market that help the software engineer in the process of FP counting. However, only a very limited number of them is able to completely automate the identification of functions and the evaluation of their complexity.

In this work, we describe how Prolog has been used to write the system FUN (FUNction point measurement) for the automated measurement of FP starting from specifications expressed in the form of an Entity Relationship (ER for short) diagram plus a Data Flow Diagram (DFD for short). In particular, we consider an integration of the two diagrams inspired to [Fuggetta et al. 88], which we call ER-DFD, in which the data stores of DFD are substituted by entities and relationships of the ER. In order to automate the measurement process, we have specialized the rules for counting FP to the case of a specification in the form of an ER-DFD. The informal counting rules expressed in natural language in [IFPUG 94a] have been translated into rigorous rules expressing properties of the ER-DFD graph.

Prolog was chosen for its declarativity, expressiveness, readability, maintainability and symbol handling facilities. Declarativity allowed a nearly direct translation of rigorous rules into code and the implementation of a working prototype in a short time. Exploiting the expressiveness of the relational representation and the uniformity of code and data, it was possible to easily represent the input ER-DFD graph with Prolog facts. Readability and ease of maintenance are important features since the FP counting method is in continual evolution. Symbol handling facilities, like unification and lists, were particularly useful since the task consists mainly in manipulating symbols. Finally, the possibility of dynamically changing the program, through asserts and retracts, allowed to avoid the recomputation of intermediate results thus obtaining effective optimizations. The system has been implemented in Sicstus Prolog version 3#5 [SICS 95] and it is available at the address

`http://www-lia.deis.unibo.it/Software/FUN/`

The paper is organized as follows. In Section 2 we describe the FP measurement process. In Section 3 we recall ER and DFD specifications and present their integration into ER-DFD specifications. Section 4 describes the application of FP rules to ER-DFD and presents their implementation in Prolog. Related works are discussed in Section 5. Conclusions and future work follows.

2 Function Point Measurement Process

FP measurement rules are defined in the International Function Point User Group (IFPUG) Counting Practices Manual [IFPUG 94a]. The method is based on identifying and counting the *functions* that the application has to provide, i.e. *Internal Logical Files* and *External Interface Files* (*data functions*), *External Inputs*, *External Outputs* and *External Inquiries* (*transactional functions*). Each function identified in the system is then classified into three levels of complexity (*simple*, *average* and *complex*), and a number of FP is assigned to each function according to the function type and the complexity. The rules for identifying the functions and for determining their complexity are expressed in natural language and they are referred to a number of high level abstractions defined in the manual [IFPUG 94a]. Rules have been kept informal and abstract so that they can be applied to any kind of description of the system, from a requirement document to an implementation of the system. However, as a consequence, they are vague to a certain extent and not completely free from ambiguities.

The sum of the FP contributions from all the functions gives the unadjusted FP count. The final FP count is then obtained by multiplying the unadjusted count by an adjustment factor that expresses the influence of 14 *general characteristics* of the system on which the application will run.
FP count is thus performed in 6 steps:

1) identifying the type of FP count: development project, enhancement project or application;
2) identifying the boundary of the application subject to the measure;
3) identifying the data functions, classified as Internal Logical Files (ILF) and External Interface Files (EIF), and evaluating their complexity by counting the number of Data Element Types (DET) and Record Element Types (RET) for each function;
4) identifying the transaction functions, classified as External Inputs (EI), External Outputs (EO) and External Inquiries (EQ) and their complexity by counting the number of Data Element Types (DET) and File Types Referenced (FTR) for each function;
5) determining the number of unadjusted FP by summing the contributions of all functions;
6) computing the final number of FP by multiplying the unadjusted FP count for the adjustment factor.

Our aim is to automate steps 3), 4) and 5) starting from the specification of the system in terms of an ER-DFD diagram. Steps 3) and 4) are the most complex, time consuming and prone to error, therefore they are the most interesting to automate. We assume to be performing an application count and that the boundary of the application is indicated in the ER-DFD diagram. We do not automate step 6 because it requires many notions on the application and on the environment that are not present in the ER-DFD specification and are difficult to formalize. Moreover, IFPUG is planning to remove it from the official counting practices because it is the most subjective step and many companies prefer to consider only the unadjusted FP count.

3 Entity Relationship - Data Flow Diagrams

We will perform the measurement on the specification of the application expressed by an Entity Relationship diagram [Chen 76] integrated with a Data Flow Diagram [DeMarco 78]. We consider an integration of the diagrams which is similar to Formal Data Flows Diagrams [Fuggetta et al. 88]: the data stores of DFD are replaced by entities and relationships of the ER diagrams, therefore we have data flows entering directly into entities and relationships and coming out from them. We call such an integrated diagram an ER-DFD. In order to distinguish between elements of the DFD and ER diagram which have a similar graphical symbol, we adopt the following conventions: external agents (the user or other applications) of DFD are represented with a dashed line box to distinguish it from entities represented as normal boxes and data flows are represented by arrows to distinguish them from the connections between entities and relationship represented as simple lines.

A number of fields are associated with each data flow: when a field has the same name of an attribute of an entity, they refer to the same data. When the field does not correspond to any attribute, it can represent control information used to influence the function performed by the process, or it can represent data derived from attributes by computation.

We suppose that the diagram contains also the indication of the boundaries of the different applications in the form of dashed lines.

3.1 Example of ER-DFD specification

We now present the ER-DFD specification of an application for the management of Human Resources. This application is the subject of a series of case studies [IFPUG 94b, IFPUG 94c, IFPUG 94d] of Function Point measurement. In particular, in [IFPUG 94c] the measurement is performed from a specification expressed as an ER diagram and a DFD. This application will be used throughout the paper in order to illustrate the FP measurement process and the behaviour of the system.

The aim of the application is to manage information about employees of a firm. In particular, the user requires to store information about each employee, comprehending data on the dependents of the employee, data on the salary or the hourly rate and data on the work location. The location must be a valid location in the application Fixed Asset. If the employee works abroad, the hourly rate must be converted to US dollars by accessing the application Currency and retrieving the conversion rate. Moreover, the application has to store information about different jobs, together with a description composed of a series of text lines. Finally, the user requires to store information about the assignment of jobs to employees. Table 1 shows the attributes of entities and relationships and Figure 1 the ER diagram[1].

Entities or relationships	**Attributes**
Employee	Social_Security_Number (key), Name, Nbr_Dependents, Type_Code.
Salaried	Supervisory_level
Hourly	Standard_Hourly_Rate, US_Hourly_Rate, Collective_Bargaining_Unit_Number
Dependent	Dep_SSN (key), Dep_name, Dep_birth_date
Job	Name, Job_Number (key), Pay_grade
Description	Line_Number, Description_Line
Job Assignment	Effective_Date, Salary, Performance_Rating
Location	Location_Name (key), Address, City, State, Zip, Country.
Conversion Rate	Conversion_Rate_To_Base_Currency.

Table 1: Attributes of entities and relationships.

The processes that the user requires are adding, changing, deleting, inquiring and reporting information about employees, jobs and job assignments. In reporting, the total number of shown entities is usually printed as well.

Among these processes, we will here consider in more details the following:

- adding an employee, together with data on his dependents and the salary or hourly wage (see Figure 2 for the addition process);
- reporting on all employees, printing the list of employees together with their total number (Figure 3);

[1] The entire ER diagram is shown separately from the processes of the DFD for clarity.

- inquiring on the data of an employee, given his social security number (Figure 4);
- adding job information, together with its description (Figure 5);
- adding a job assignment (Figure 6).

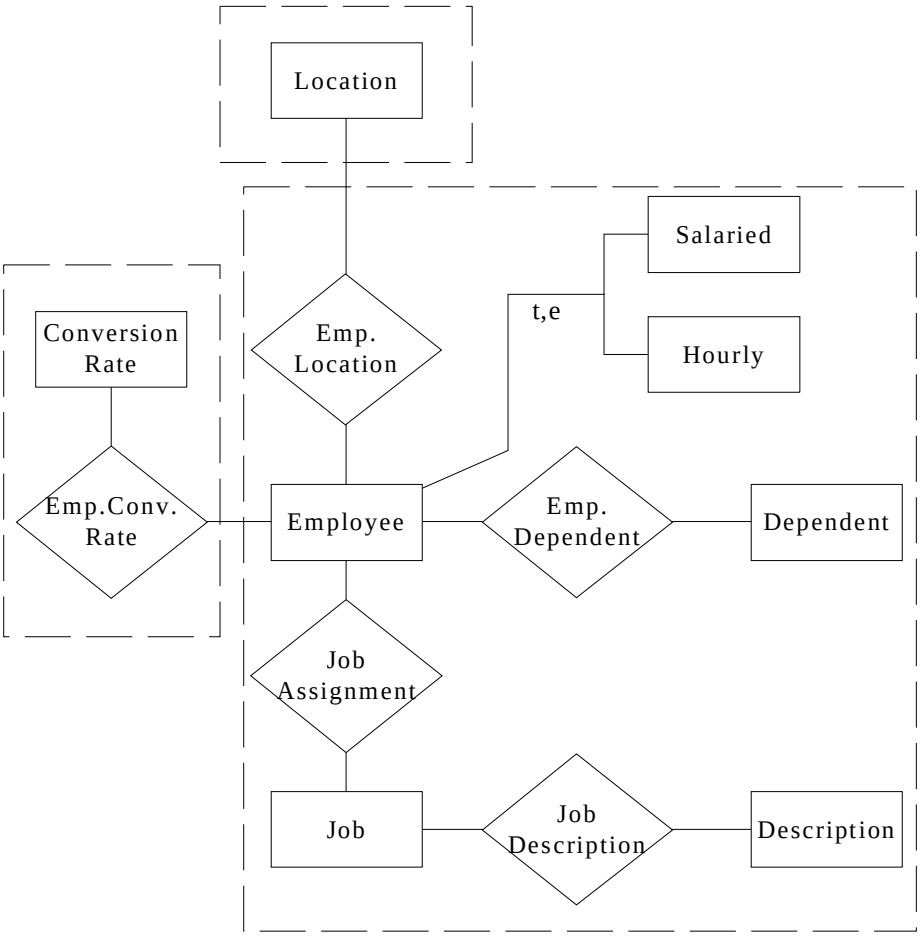

Figure 1: Complete ER diagram for the Human Resource application.

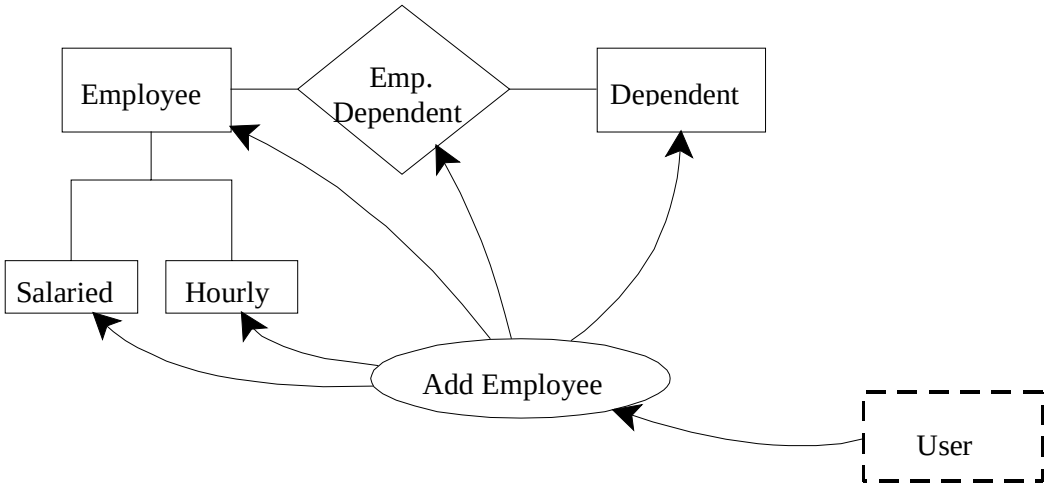

Figure 2: Add Employee process.

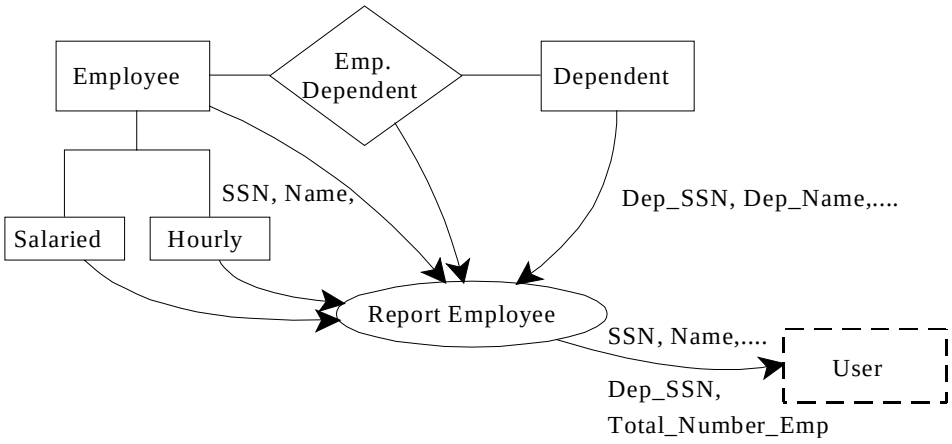

Figure 3: Report Employee process.

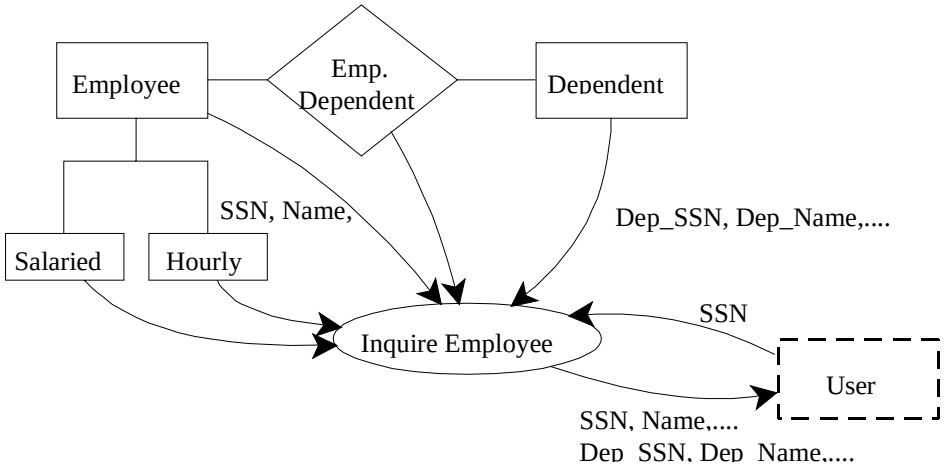

Figure 4: Inquire Employee process.

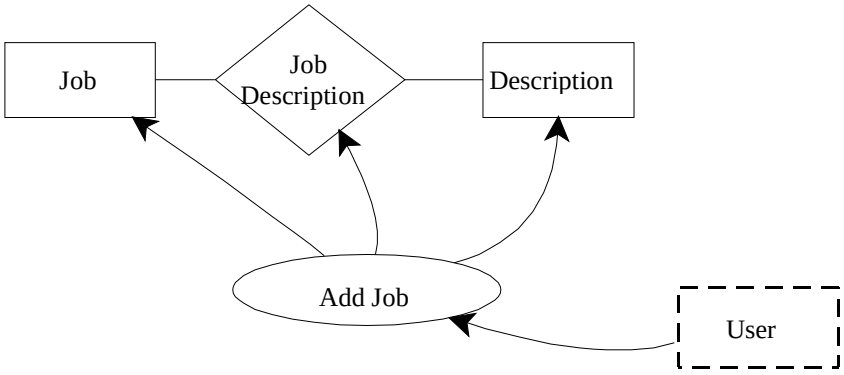

Figure 5: Add Job process.

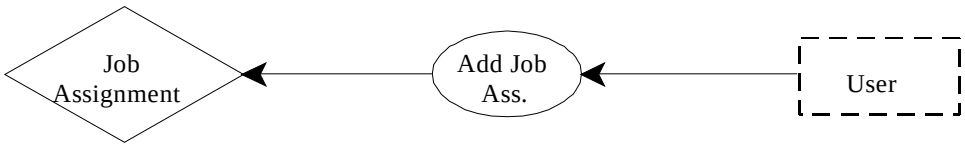

Figure 6: Add Job Assignment process.

3.2 Prolog Representation of ER-DFDs

We represent ER-DFDs in Prolog with facts of the form:

- `application(name,[ent1,..,entn],[rel1,..,reln],[proc1,..,procn).`

where `name` is an application containing entities `ent1,...,entn`, relationships `rel1,...,reln` and processes `proc1, ...,procn`.

- `entity(name,[attrib1,...,attribn]).`

where `name` is an entity with attributes `attrib1,...,attribn`. We do not record which attribute is the entity key because we do not need this information in the FP count.

- `specialization(name,[child1,...,childn],total,exclusive).`

for stating that entity `name` has sub-entities `child1,...,childn` and `total` and `exclusive` can be replaced by the Boolean constants `0` and `1` stating whether the hierarchy is total and/or exclusive.

- `relationship(name,e1,e2,[attrib1,..,attribn],key_e1,key_e2]).`

where `name` is a binary relationship between `e1` and `e2` with attributes `attrib1,..,attribn`. `key_e1` is the key of `e1`, `key_e2` the key of `e2`. `key_e1` (`key_e2`) is replaced by the constant `nil` if `key_e1` (`key_e2`) is not present in the relational implementation of `e2` (`e1`). This kind of information should not appear in a specification document but it is necessary to perform the count. This is a case were the FP count is not completely implementation independent.

- `dataflow(sorg,dest,[field1,...,fieldn]).`

To indicate that there is a data flow from `sorg` to `dest` with fields `field1,...,fieldn`.

4 Rules for Counting Function Points from ER-DFD

In this section we present rules for counting Function Points from the specification of an application expressed in the form of an ER-DFD. For each function, we have translated IFPUG informal rules into rigorous rules expressing properties of the ER-DFD graph. The rules obtained are rigorous because all the ambiguities and vagueness of IFPUG rules have been removed. It was thus possible to translate them into Prolog code. Thanks to Prolog's declarativity, the translation was straightforward.

For Internal Logical Files and External Inputs, both IFPUG definition and identification rule are presented in full details, so that the reader can compare them with the rigorous ones. For brevity, only IFPUG definitions have been reported for the other functions and for the complexity of Data Element Types, Record Element Types and File Types Referenced, apart from some cases when the rule was needed for clarity.

In Sections 4.1 and 4.2, we discuss identification and complexity rules for data functions. In Sections 4.3 and 4.4, we describe identification and complexity rules for transactional functions.

4.1 Identification Rules for Data Functions

Data functions are Internal Logical Files (ILF for short) and External Interface Files (EIF).

4.1.1 Internal Logical Files

The IFPUG definition of an ILF is:

> *"an ILF is a group of logically related data or control information, user recognizable, maintained inside the application boundary"* [IFPUG 94a].

Let us now introduce the terminology used in IFPUG's manual.

> "Control information *are data used by the application to maintain the conformity with the functional requirements specified by the user"* [IFPUG 94a].
>
> *"The term* user recognizable *... refers to specific user requirements that an expert user would define for the application"* [IFPUG 94a].
>
> *"The term* maintained *refers to the capacity of modifying the data with an elementary process"* [IFPUG 94a].
>
> *"An* elementary process *is the smallest unit of meaningful activity from the user perspective.... it is self contained and leaves the application in a consistent state"* [IFPUG 94a].

The identification rule for ILFs is: a group of data or control information is an ILF if it satisfies all the following conditions:

1) *"The group of data or control information is a logical, or user identifiable, group of data that fulfills specific user requirements.*
2) *The group of data is maintained within the application boundary.*
3) *The group of data is modified, or maintained, through an elementary process of the application.*
4) *The group of data identified has not been counted as an EIF for the application."* [IFPUG 94a].

Let us now express in a rigorous way the application of the IFPUG identification rule to ER-DFD. Groups of logically related data are represented by sets of connected entities and relationships while elementary processes are represented by processes in the DFD. We make the assumption that every process in the DFD is an elementary process. This assumption is reasonable if we consider a full detailed DFD where it is not possible to further decompose the processes.

We need as well the definition of the term*maintained*.

Definition 1 (Maintained) An entity or a relationship is maintained by a process if and only if there is a data flow coming from the process and entering in the entity or relationship.

The ILF identification rule for ER-DFD is:

Rule 1 (ILF identification) A set of connected entities and relationships is an ILF if:

1) **all the elements of the set are inside the application boundary;**
2) **there is at least one process of the application that maintains all the elements in the set and no elements outside it.**

For example, in the Human Resources application, {Employee, Salaried, Hourly, Employee-Dependent, Dependent} is an ILF because it is inside the application boundary and its elements are maintained through the elementary process Add Employee: it is not possible to add an Employee without adding also his Dependent employees and therefore they must be put together in a single

logical file. Other ILFs are {Job, Description, Job Description} and {Job Assignment} because they are inside the application boundary and they are maintained, respectively, by the processes Add Job and Add Job Assignment.

Now that IFPUG rules have been made rigorous and unambiguous, and it is now possible to translate them into Prolog code. The predicate `ilf/2` is used to identify ILFs: `ilf(Appli,ILF)` succeeds if `ILF` is a list containing the entities and relationships of an ILF for the application `Appli`. Rule 1 above is implemented by the following Prolog clause (in Sicstus Prolog syntax):

```
ilf(Appli,ILF):-
   application(Appli,EntList,RelList,ProcList),
% pick a process Proc
   member(Proc,ProcList),
% find the ent. and rel. that are maintained by Proc
   findall(ER,dataflow(Proc,ER,_),ILF),
% verify that ILF is inside the boundary of Appli,
   append(EntList,RelList,ERList),
   sublist1(ILF,ERList),
% that is not empty, and connected
   ILF \== [],
   connected(ILF).
```

We use the Sicstus built-in predicate `findall(Template,Goal,Bag)` that assigns to `Bag` a list of instances of `Template` in all proofs of `Goal` found by Prolog. All variables in `Goal` are taken as being existentially quantified. The call `findall(ER,dataflow(Proc,ER,_), ILF)` returns in `ILF` the list of all the entities and relationships maintained by the process `Proc`. The predicate `sublist1(Sublist,List)`[2] verifies that all elements of `Sublist` are in `List`. For brevity we omit the definition for`connected/1`.

4.1.2 External Interface Files

The IFPUG definition of an EIF is:

> *"an EIF is a group of logically related data or control information, user identifiable, referenced by the application but maintained inside the boundary of another. This means that an EIF counted for an application must be an ILF for another"* [IFPUG 94a].

Let us now present the rigorous identification rule. We first have to define the term*referenced.*

Definition 2 (Referenced) An entity or a relationship is referenced by a process if and only if there is a data flow from the entity or relationship to the process.

The EIF identification rule for ER-DFD is:

Rule 2 (EIF identification) A set of connected entities and relationships is an EIF if:

1) **all the elements of the set are inside an external application;**
2) **the set satisfies the condition for ILFs with respect to the external application;**
3) **at least one element of the set is referenced by a process inside the counted application;**
4) **no element of the set is maintained by processes inside the counted application.**

---

[2] It was called `sublist1/2` in order to distinguish it from the Sicstus list library predicate `sublist/2` that performs a different computation.

In Figure 7 we show a more detailed example of an EIF. The EIF is composed of two entities and a relationship that are outside of the counted application boundary and satisfy the ILF rule for the external application. Moreover, the set has a data flow going from one of its entities to a process in the counted application. It is important to note that it is not necessary to have data flows from all the elements of the set. Finally, no data flow enters in the set from the counted application.

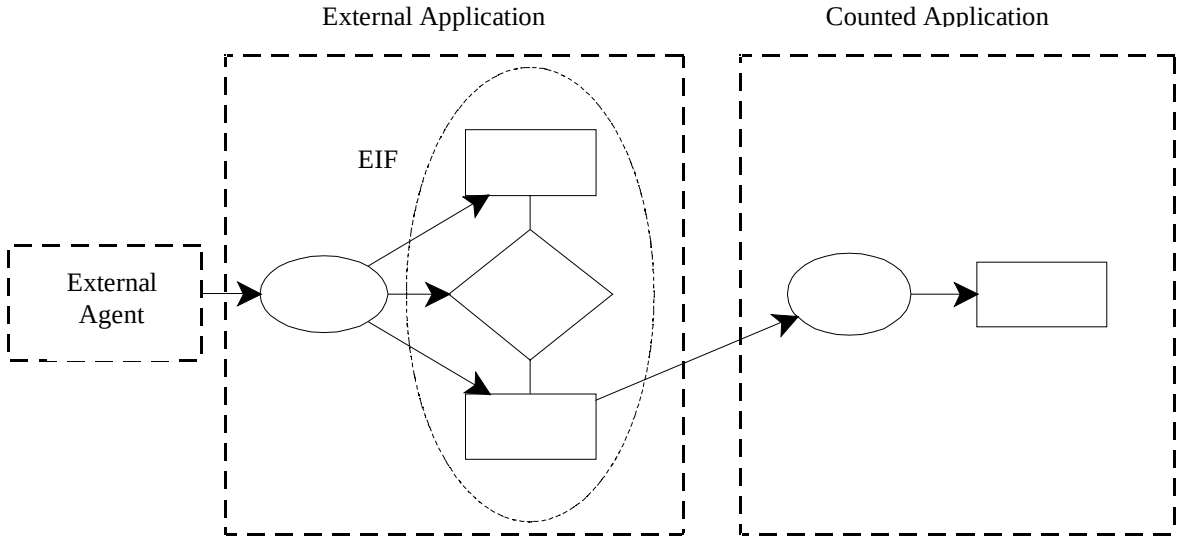

Figure 7: Example of EIF

In the Human Resources application,. we are supposed to be given the information that {Location} is an ILF for the application Fixed Assets and {Conversion Rate} is an ILF for the application Currency, since we do not know the processes for these applications. Both {Location} and {Conversion Rate} are EIFs because they are referenced but not maintained from the counted application.

Predicate `eif/2` is used to identify EIFs. Rule 2 above is implemented by the following Prolog clause:

```
eif(Appli,EIF):-
% find an external application Appli1
   application(Appli1,_,_,_),
   Appli1 \== Appli,
% find an ILF for Appli1
   ilf(Appli1,EIF),
% an element of EIF is referenced by a process in Appli ?
   member(Ent,EIF),
   dataflow(Ent,Proc,_),
   application(Appli,_,_,ProcList),
   member(Proc,ProcList),
% check that EIF is not maintained by any process of Appli
   \+ maintained(EIF,Appli).
```

`\+` is the Sicstus operator for negation. Note that at this point the calls to `ilf/2` would repeat the identification, since the information about identified files are already asserted in the database.

The predicate `maintained(EIF,Appli)` verifies that there is a dataflow from a process of `Appli` to an element of `EIF`.

4.2 Complexity Rules for Data Functions

In order to assign the right number of FP to each identified data function, we have to count the Data Element Types and Record Element Types associated with the function.

> *"A Data Element Type (DET) is a unique field, user recognizable and non recursive on an ILF or EIF"* [IFPUG 94a].

A *recursive field* is an external key that is used in the relational implementation to establish a relationship between two entities in the same logical file. Recursive fields are not counted as DETs because otherwise the same attribute would be counted twice for the logical file, once as an entity attribute and once as an external key. Instead, we count as DETs external keys pointing to entities outside the file. As mentioned in Section 3, this is a case where the FP count is not completely independent from implementation details, since we need to have information about the relational design.

We make the assumption that every attribute of an entity or a relationship is unique, user recognizable and not recursive, i.e., it is a DET. This assumption is not very restrictive, if the ER diagram has been correctly laid down.

Rule 3 (DET counting) Count:

1) **one DET for each attribute of the entities and relationships in the logical file;**
2) **one DET for each attribute composing the keys of entities in other files connected by relationship to entities of the file (external keys), unless the relationship constitute a file (in this case the keys of the entities connected by the relationship are stored only in that file).**

In order to count DETs, each entity and relationship of the logical file is considered in turn, and DETs are counted for it. Then all counted DETs are summed to give the total for the file.

In the Human Resources application, the ILF {Employee, Salaried, Hourly, Employee-Dependent, Dependent} has 11 DETs, one for each attribute of its entities, plus 2 DETs for the external keys, one for Location and one for Conversion Rate. {Job, Description, Job-Description} has 5 DETs: 5 DETs for the five attributes of Job and Description. {Job Assignment} has 5 DETs: 3 DETs for the three attributes of the relationship and 2 DET for the external keys Social_Security_Number (link to Employee) and Job_Number (link to Job). These external keys are not counted as DETs for, respectively, {Job, Description, Job-Description} and {Employee, Salaried, Hourly, Employee-Dependent, Dependent}, because the relationship connecting them, Job Assignment, constitute an ILF.

Predicates `entDet/3` and `relDet/3` are used to count DETs respectively for entities and relationships in the file: `entDet(Name,DET,File)` returns in `DET` the number of DETs of entity `Name` belonging to `File` (and similarly for `relDet/3` where `Name` is a relationship). They compute the DET number by summing the number of attributes of the entity or relationship with the number of external keys. The number of external keys are given by the two predicates `dets_from_external_keys_ent/3` and `dets_from_external_keys_rel/3`. We show the code of one of them as an example:

```
dets_from_external_keys_ent(H,File,Det):-
% finds all external relations that connect entity H with
% an external entity whose key must be stored in File.
% Consider 2 cases depending on the position of H
% as an argument of Rel
   findall([Rel,Key],
        (relation(Rel,H,_,_,[_,Key],_,_,_,_),Key\==nil,
        \+member(Rel,File),
% [Rel] must not be a file otherwise the external keys
% are stored only in the file (see Job Assignment)
        \+ file([Rel])),
        ExternalRel1),
   length(ExternalRel1,L1),
   findall([Rel,Key],
        (relation(Rel,_,H,_,[Key,_],_,_,_,_),Key\==nil,
        \+member(Rel,File),\+ file([Rel])),ExternalRel2),
   length(ExternalRel2,L2),
   Det is L1 + L2.
```

The IFPUG definition for RETs is:

"A Record Element Type (RET) is a user identifiable subgroup of data elements in an ILF or EIF" [IFPUG 94a].

The IFPUG counting rule is:

"Count one RET for each optional or mandatory subgroup of an ILF or EIF.
If there are no subgroups, count the ILF or EIF as one RET" [IFPUG 94a].

When the user adds a new group of data to an ILF or EIF, she/he must add at least one of the *mandatory subgroups* and zero, one or all *optional subgroups*. Subgroups consist of one or more subentities, depending on the type of hierarchy. In order to illustrate how subgroups are composed, let us consider the case of an entity with two subentities. We have four cases.

1) Exclusive and total hierarchy: we have 2 mandatory RETs, one consisting of the attributes of the parent entity plus the attributes of one sub-entity and the other consisting of the parent plus the other sub-entity.
2) Exclusive and not total hierarchy: we count 3 RETs, 1 mandatory for the parent entity and 2 optional for each of the sub-entities, because an instance of the parent may not be in any of the children.
3) Not exclusive and total hierarchy: we count 3 mandatory RETs, one for the parent entity plus one child, one for the parent entity plus the other child and one for the parent entity plus both children.
4) Neither total nor exclusive hierarchy: we count 1 mandatory RET for the parent and 2 optional RETs for the children.

We can now give the rule for counting RETs on ER-DFDs.

Rule 4 (RET counting) Count one RET for each entity and for each relationship with attributes of the logical file. If an entity has sub-entities, then count RETs for each sub-entity, sum the results over all sub-entities and add 1 if the hierarchy is not both total and exclusive.

As for DETs, we count RETs for each entity and relationship separately and then we sum the results.

In the Human Resources application, the ILF {Employee, Salaried, Hourly, Employee-Dependent, Dependent} has 2 RETs for Employee and its two sub-entities, since the hierarchy is total and exclusive, 1 RET for Dependent and no RET for the relationship Employee-Dependent because it does not have any attribute. {Job, Description, Job Description} has 2 RETs: 1 RET for Job, 1 for Description. {Job Assignment} has 1 RET.

Predicates `entRet/2` and `relRet/2` are used to count RETs respectively for entities and relationships in the file: `entRet(Name,RET)` returns in `RET` the number of RETs of entity `Name` (and similarly for`relRet/2`).

```
% case where Name has sub-entities
entRet(Name,Ret):-
   specialization(Name,ChildList,Escl,Tot),!,
% count RETs for the sub-entities
   entRet_rec(Retch,ChildList,0),
% compute RETs for Name according to the hierarchy type
   gerarch(Ret,Retch,Escl,Tot).

% case where Name has not sub-entities: count 1 RET
entRet(_Name,1).

entRet_rec(Ret,[],Ret):-!.
entRet_rec(Ret,[Head|Tail],Acc):-
   entRet(Head,Ret1),
   Acc1 is Acc + Ret1,
   entRet_rec(Ret,Tail,Acc1).

% add 1 if the hierarchy is not both total and exclusive
gerarch(Ret,Ret,1,1):-!.
gerarch(Ret,Retch,_,_):-
   Ret is Retch + 1.

% counts RETs for relationships
relRet(Name,Ret):-
   relation(Name,_,_,FieldList,_,_,_,_,_),
   length(FieldList,L),
   (L>0 ->   Ret=1;
        Ret=0
   ).
```

Once the number of DETs and RETs for ILFs and EIFs have been determined, it is possible to compute the number of function points to be assigned to each function. For each function type, a complexity matrix gives the function complexity in terms of the number of DETs and RETs. The complexity can assume the values: low, average and high. From the value of the complexity, a conversion table gives the number of function points. Table 2 shows the *complexity matrix* for ILFs and EIFs, Table 3 shows the conversion table for ILFs and Table 4 the conversion table for EIFs.

	1-19 DET	20-50 DET	51 or more DET
1 RET	low	low	average
2-5 RET	low	average	high
6 or more RET	average	high	high

Table 2: Complexity matrix for ILFs.

Complexity	Function Points
low	7
average	10
high	15

Table 3: Conversion table for ILFs.

Complexity	Function Points
low	5
average	7
high	10

Table 4: Conversion table for EIFs.

4.3 Identification Rules for Transactional Functions

Transactional Function are classified into External Inputs, External Outputs and External Inquiry.

4.3.1 External Inputs

> *"An External Input (EI) elaborates data or control information coming from outside the application boundary. The EI is itself an elementary process. Elaborated data maintain one or more ILFs. Control information may or may not maintain an ILF"* [IFPUG 94a].

We have to distinguish between EI of data and EI of control information. The IFPUG rule states that the conditions that must be satisfied by a process for it to be an EI of data are:

1) *"Data is received from outside the application boundary.*
2) *Data in an ILF is maintained through an elementary process of the application.*
3) *Process is the smallest unit of meaningful activity from the user perspective.*
4) *Process is self contained and leaves the business in a consistent state.*
5) *For the identified process, one of the following two rules apply:*
   I. *Processing logic is unique from other external inputs.*
   II. *Data elements are different from other external inputs."* [IFPUG 94a].

The conditions that must be satisfied by a process in order to be an EI of control information are:

1) *"Control information are received from outside the application boundary.*
2) *Control information are specified by the user to ensure the conformity with the functional requirements.*
3) *For the identified process, one of the following two rules apply:*
   I. *Processing logic is unique from other external inputs.*
   II. *Data elements are different from other external inputs."* [IFPUG 94a].

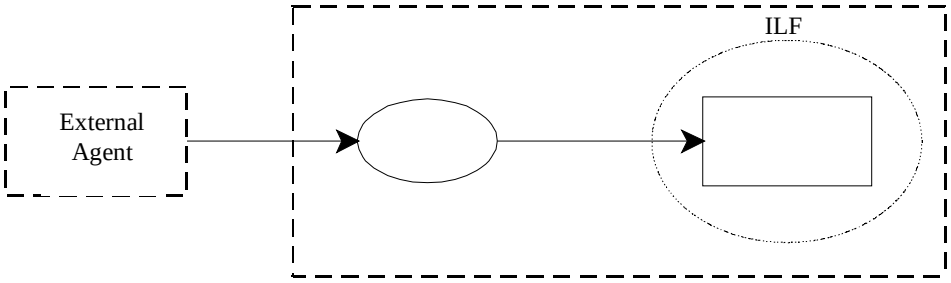

Figure 8: Simplest case of an EI of data

The simplest case of EI of data is shown in Figure 8.
In order to identify EIs in the ER-DFD diagram, we have to consider the processes in the diagram. We have assumed that all processes in the DFD are elementary processes, therefore conditions 2) and 3) for EIs of data are satisfied. Moreover, we assume that the processing logic of every process in the DFD is unique, thus satisfying condition 5-I. for EIs of data (condition 3-I. for EIs of control information). This assumption is not very restrictive since it is reasonable for a DFD diagram not to have duplicated processes. We did not consider the case of processes with the same processing logic but different type of data as input, since in our opinion it is difficult to conceive how two identical processes can elaborate different data.

A process in the ER-DFD is an EI of data if the following conditions are satisfied:

1) It has at least one data flow from outside the application boundary.
2) It has at least one data flow entering in an ILF.

A process in the ER-DFD is an EI of control information if it has at least one data flow from outside the application. Thus, the only difference between identification rules for EIs of data and control information is that EIs of control information may or may not maintain an ILF. Therefore, to identify both kinds of EIs, it would be enough to check that the process has at least one data flow from the outside. However, also the other transactional functions may have data flows from the outside, therefore, in order to distinguish EIs form EOs and EQs, we require that EIs must not have any data flow going outside the application boundary.

Rule 5 (EI identification) A process of the ER-DFD belonging to the application is an EI if

1) there is at least one data flow from the outside to the process,
2) there is no data flow from the process to the outside.

In the Human Resources application, processes Add Employee, Add Job Assignment and Add Job are EIs because there are data flows to them from the outside and there are no data flows from them to the outside.

In order to translate Rule 5 above into Prolog code, we use two predicates `dataflow_from_outside(Appli,Proc)` and `dataflow_to_outside(Appli, Proc)`. As an example, we report the code of the first:

```
dataflow_from_outside(Appli,Proc):-
% pick a data flow to Proc
   dataflow(Source,Proc,_),
% check that Source is outside Appli
   application(Appli,EntList,RelList,_),
   append(EntList,RelList,ERList),
   \+ member(Source,ERList).
```

4.3.4 External Inquiry

"An External Inquiry (EQ) is an elementary process composed by a combination of input and output that results in data retrieval. The output side does not contain derived data. No ILF is maintained during the process" [IFPUG 94a].

Figure 9 shows the simplest case of EQ.

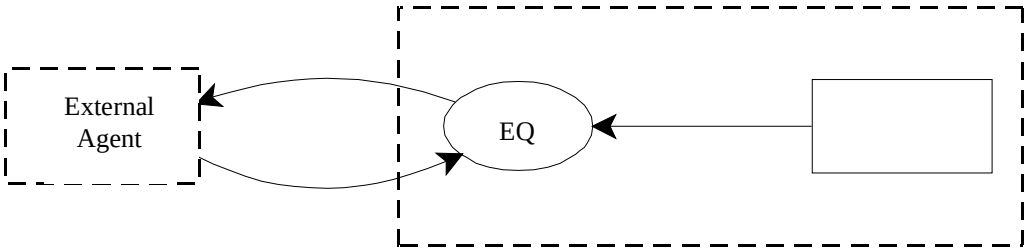

Figure 9: Simplest case of an EQ.

Rule 6 (EQ identification) A process of the ER-DFD is an EQ if

1) **there is at least one data flow from the outside to the process (input part);**
2) **there is at least one data flow from the process to the outside (output part);**
3) **there is at least one data flow from an ILF to the process;**
4) **all the fields of data flows going outside the application boundary are among the fields of data flows to the process.**

In the Human Resources application, the process Inquire Employee is an EQ.

To check that no derived data are present in the dataflows going outside the application boundary, the predicate `no_derived_data(Proc)` is used in the definition of `eq(Appli,Proc)`.

```
no_derived_data(Proc):-
    findall(Fields,dataflow(_From,Proc,Fields),FListsIn),
    appendall(FListsIn,FieldsIn),
    findall(Fields,dataflow(Proc,_To,Fields),FListsOut),
    appendall(FListsOut,FieldsOut),
    sublist1(FieldsOut,FieldsIn).
```

The predicate `appendall(List_of_lists,List)` returns in `List` the concatenation of all lists in `List_of_lists`.

4.3.2 External Output

"An External Output (EO) is an elementary process that generates data or control information that are sent outside the application boundary" [IFPUG 94a].

Figure 10 shows the simplest case of an EO.

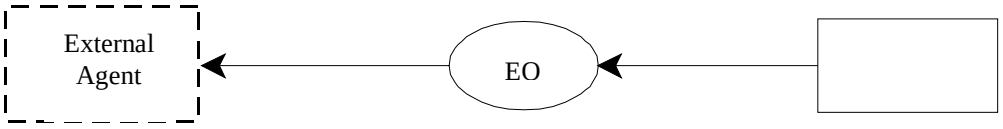

Figure 10: Simplest case of EO.

What distinguishes an EO from an EQ is the fact that an EQ does not elaborate the retrieved data, while an EO outputs derived data. Therefore we have the following rule for ER-DFD.

Rule 7 (EO identification) A process in the ER-DFD is an EO if

1) **there is at least one data flow from an ILF to the process;**
2) **there is at least one data flow from the process to the outside;**
3) **data flows from the process to the outside contain at least one field that is not contained in any of the data flows from ILFs to the process.**

In the Human Resources application, the process Report Employee is an EO because there the data flow from it to the outside contains the field Total_Number_Emp that is not in the flows from the ILF {Employee, Salaried, Hourly, Employee-Dependent, Dependent}.

To check condition 3), a call to `\+ no_derived_data(Proc)` is used in the definition of `eo(Appli,Proc)`.

4.4 Complexity Rules for Transactional Function

In order to assign the right number of FP to each identified transactional function, we have to count DETs and File Types Referenced associated with the function. The definition for DETs is the same as the one for data functions.

A File Type Referenced is

- an Internal Logical File read or maintained by an EI or read by an EO or EQ,
- an External Interface File read by the function.

Rule 8 (EI, DET counting) Count one DET for each field in data flows from the EI to internal entities and relationships.

```
eiDet(Proc,Det):-
   findall(Fields,dataflow_to_int(Proc,Fields),FList),
   appendall(FList,FieldsDup),
   remove_duplicates(FieldsDup,Fields),
   length(Fields,Det).

dataflow_to_int(Proc,Fields):-
   application(_Appli,Ent,Rel,ProcL),
   member(Proc,ProcL),
   dataflow(Proc,Des,Fields),
   (member(Des,Ent);member(Des,Rel)).
```

Rule 9 (EI, FTR counting) Count one FTR for each ILF maintained by the process and one FTR for each ILF or EIF referenced by the process.

To implement this rule, we have used a predicate `eif_ilf_referenced(Proc, File)` that returns a file referenced by the process:

```
eif_ilf_referenced(Proc,File):-
% find the application containing Proc
   application(Appli,_,_,ProcList),
   member(Proc,ProcList),
% find the Source of a dataflow to Proc
   dataflow(Source,Proc,_),
% Source belongs to an ILF or EIF
   (ilf(Appli,File);eif(Appli,File)),
   member(Source,File).
```

Note that at this point ILFs and EIFs are already known and the calls to `ilf/2` and `eif/2` are not repeated since information about identified files have been asserted in the database.

For EQ, we have to distinguish between the input side and the output side. We consider the input side as the data flows from the user to the EQ, while the output side is composed of the data flows from the files to the EQ and from the EQ to the user.

Rule 10 (EQ, input side, DET counting) Count one DET for each field in the data flows from the outside to the process.

Rule 11 (EQ, input side, FTR counting) Count one FTR for each ILF that has at least one DET among the fields of the data flow from the outside to the EQ.

To implement this rule, we have used a predicate `file_input_side(Proc,File)` that finds an internal `File` whose attributes appear in the data flow from outside to `Proc`:

```
file_input_side(Proc,File):-
    dataflow(Source,Proc,Fields),
    application(Appli,E,R,P),
    member(Proc,P),
    \+ member(Ent,E),
    \+ member(Ent,R),
    ilf(Appli,File),
    file_fields(File,ILFFields),
    intersection(Fields,ILFFields).
```

Rule 12 (EQ output side, DET and FTR counting) Count DETs and FTRs exactly as for EOs.

Rule 13 (EO, DET counting) Count one DET for each field of the data flows from the process to outside.

Rule 14 (EO, FTRs counting) Count one FTR for each ILF or EIF referenced by the process.

As for data functions types, once the number of DETs and FTRs for each transactional function have been determined, it is possible to compute its function points. For EIs and EOs, two different complexity matrixes give the function complexity in terms of the number of DETs and FTRs. Then, the complexity is translated into function points with the conversion tables for EIs and EOs. For EQs, separate complexities for the input side and the output side are determined using, respectively, the matrixes of EIs and of EOs. Then the highest complexity is chosen and it is translated into function points using the conversion table for EQs. For Complexity matrixes and conversion tables for transactional functions see [IFPUG 94a].

6 Related Works

[Mendes et al. 96] is a survey of FP tools available on the market. The authors review 52 tools, out of which only 8 perform automatic FP counting. They are shown in Table 5.

Tool	Vendor
Autopoint	Integrated Software Specialists
Before You Leap v. 1.52	Strategic Systems Technology Ltd.
Composer FP Report	Texas Instruments
FP Analyst	Cayenne Software Inc.
LDA - LINC Development Assistant	Unisys
Revolve v. 3.1	Micro Focus Ltd.
PCA Calc Add-on (prototype)	System House SHL Québec
VIA RECAP: ESW Portfolio Analysis	VIASOFT Inc.

Table 5: FP counting tools.

Among these tools, 3 perform the counting starting from ER, DFD or similar diagrams: *FP Analyst* takes as input Cayenne models that contain DFD, *Composer FP Report* starts from ER diagram plus Dialog Flow / Process Action diagrams, *Before You Leap* starts from DFD and ER-D. All these tools perform identification of the functions and evaluation of their complexity, except for *Before You Leap* that is not able to count DETs and RETs for data function types. Moreover *FP Analyst* and *Composer FP Report* are also able to automatically identify the application boundary and *FP Analyst* is able to decide which files and processes are unique, while we consider all files and processes as unique by assumption. However, to the best of our knowledge, FUN is the only system for FP measurement written in Prolog. In this way, we obtain a system that is easy to maintain, allowing rules to be easily modified as new versions of the counting rules are released. Moreover, our analysis, besides leading to a counting tool, has the benefit of increasing the understanding of the FP rules when counting from ER-DFD, removing the ambiguities that are present in the counting rules and translating them into simple rigorous rules by making only a limited number of assumptions.

7 Conclusions and future works

In this paper we have presented the system FUN for the automatic counting of the Function Point metric starting from an Entity Relationship - Data Flow Diagram, a formalism that integrates ER and DFD by replacing the data stores of DFD with the entities and relationships of ER. The FP counting rules have been specialized for the case of ER-DFD and made rigorous by making a number of simplifying assumptions. Prolog was chosen for the implementation for its declarativity, expressiveness, readability, maintainability and symbol handling facilities. The translation of the rigorous rules into Prolog code was straightforward.

Rigorous FP rules also suggest a different implementation, where, besides the declarativity and symbol manipulation capabilities of Prolog, constraint propagation is also exploited. In particular, the FP counting problem might be also viewed as a Constraint Logic Programming application on Finite Domains. In this case, variables would represents data and transactional functions, the former ones initially ranging on the powerset of the set of entities and relationships, the latter ones on the set of processes. Identification rules would then be mapped into constraints among these variables. Constraint propagation thus performs the pruning of the initial domains till only identified data and transaction functions remain. This kind of implementation is subject for future work.

In the future, we also intend to apply the system to a number of cases and compare the results with those obtained by human experts. Moreover, work will be devoted to the development of a graphical user interface in which the user will draw the ER-DFD and the system will

Bibliography

[Albrecht 79] A. Albrecht. *Measuring application development productivity.* in *Proc. Joint SHARE/GUIDE/IBM Applications Development Symposium*, Monterey, CA, 1979

[Albrecht, Gaffney 83] A. Albrecht, J. Gaffney: *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*; in IEEE Trans. Software Eng., 9(6), 1983, pp. 639-648

[Chen 76] P.P. Chen. *The Entity-Relationship model. Toward a unified view of data*. ACM Transactions On Database System, Vol. 1, No. 1, Marzo, 1976

[DeMarco 78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.

[Fenton 94] N. Fenton, *Software Measurement: a Necessary Scientific Basis*, IEEE Trans. Software Eng., 20, 1994, pp. 199-206.

[Fuggetta et al. 88] A. Fuggetta, C. Ghezzi, D. Mandrioli, A. Morzenti, "VLP: a Visual Language for Prototyping", IEEE Workshop on Languages for Automation, College Park, MD, August 1988.

[IFPUG 94a] International Function Points User Group, *Function Point Counting Practices Manual, Version 4.0*, 1994.

[IFPUG 94b] International Function Points User Group, *Function Point Counting Practices: Case Studies, Case Study 1, Release 1.0*, 1994.

[IFPUG 94c] International Function Points User Group, *Function Point Counting Practices: Case Studies, Case Study 2, Release 1.0*, 1994.

[IFPUG 94d] International Function Points User Group, *Function Point Counting Practices: Case Studies, Case Study 3, Release 1.0*, 1994.

[Mendes et al. 96] O. Mendes, A Abran, P. Bourque, *Function Point Tool Market Survey 1996*, Research Report, Software Engineering Management Laboratory, Université du Quebec à Montreal, 14 December 1996.

[SICS 95] *SICStus Prolog User Manual, Release 3#0*, Swedish Institute of Computer Science, 1995.